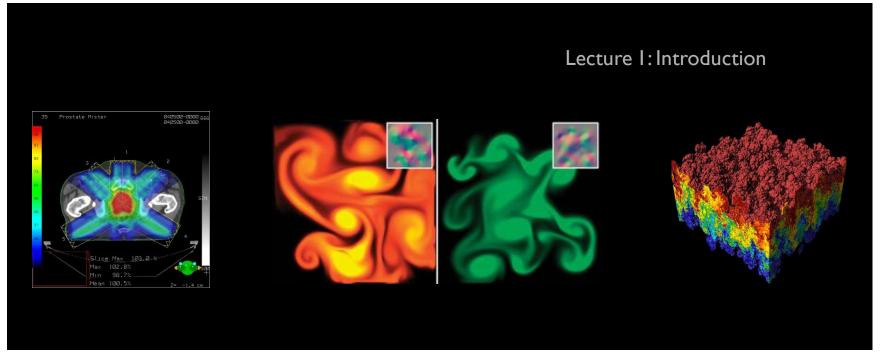
CS 179: Introduction to GPU Programming.



Administration

Main topics covered in course:

- (GP)GPU computing/parallelization
- C++ <u>CUDA</u> (parallel computing platform)

TAs:

- Ethan Jaszewski (<u>ejaszews@caltech.edu</u>)
- Ivy Tang (<u>itang@caltech.edu</u>)
- Thomas Barrett (<u>tbarrett@caltech.edu</u>)
- Alvin On (<u>aon@caltech.edu</u>)

Overseeing Instructor:

Al Barr (barr@cms.caltech.edu)

Primary Websites:

- http://courses.cms.caltech.edu/cs179/
- http://www.piazza.com/caltech/spring2021/cs179
 - Piazza is the primary forum for the course! Make sure you're enrolled!
- Also CS179 Canvas Calendar for Zoom links.

Class time:

- MW(F) 3pm PDT for Zoom classes and HW recitations.
- Also TA office hours through Zoom (TBA)

Course Requirements

Please fill out survey on Piazza for the when2meet link coming up on Piazza, for desired TA office hours

Homework:

- 6 assignments, perhaps in 5 weeks due to COVID-19
- Each worth 10% of grade
- Also "enough" work before Add Day, to pass!

Final project:

- 4-week project
- 40% of grade total

P/F Students must receive at least 60% on every assignment AND the final project.

Homework

Due on Tuesdays 3PM PDT.

First set is due Tuesday April 6th

• Use zip on remote GPU computer at Caltech to submit HW.

Collaboration policy:

- Discuss ideas and strategies freely, but all code must be your own
- Do NOT look up prior years solutions or reference solution code from github without prior TA approval
- Use additional backup method for work, on a different computer.
- Make your github repository *Private*!

Office Hours: Will be interactive, through Zoom.

Times:TBA, based on survey. Survey timezone is in PDT.

Extensions

- Ask a TA for one if you have a valid reason
- See main website for details.

Your GPU Project

Project can be a topic of your choice

We will also provide many options

Teams of up to 2 people

2-person teams will be held to higher expectations

Requirements

- Project Proposal
- Progress report(s) and Final Presentation
- More info later...

Caltech Machine and your accounts now available.

The Primary GPU machine is set up and available

- You should have received a user account in email.
- Please test access and change your password.
- GPU-enabled machine is on the Caltech campus.
- Let us know if you have problems!
- You'll be submitting HW on this computer.

Secondary CMS GPU machines are too old and are no longer operational.

Alternative GPU Machines

Alternative: Use your own machine.

You will still have to submit HW on the Caltech GPU machine.

- Must have an NVIDIA CUDA-capable GPU
 - At least Compute 3.0
- Virtual machines generally won't work
 - Exception: Machines with I/O <u>MMU virtualization</u> and certain GPUs
- Special requirements for:
 - Hybrid/Optimus systems (laptops)
 - Mac/OS X (probably no longer supported?)

Setup guide on the website is likely outdated. Can follow NVIDIA's posted installation instructions (linked on page). <u>Ubuntu 20.04</u> or later ay be easiest to install!

The CPU

The "Central Processing Unit" Traditionally, applications use <u>CPU</u> for primary calculations

- General-purpose capabilities, mostly sequential operations
- Established technology
- Usually equipped with 8 or fewer, powerful cores
- Optimal for some types of concurrent processes but not large scale <u>parallel computations</u>



The GPU

The "Graphics Processing Unit"

Relatively new technology designed for parallelizable problems

- Initially created specifically for graphics
- Became more capable of general computations

Very fast and powerful, computationally

Uses lots of electrical power



GPUs – Some of the Motivation

```
Raytracing:

for all pixels (i,j) in image:

From camera eye point,

calculate ray point and direction in 3d space if ray intersects object:

calculate lighting at closest object point store color of (i,j)

Assemble into image file
```

Each pixel could be computed simultaneously, with enough parallelism!

SIMPLE EXAMPLE

Add two arrays, A and B to produce array C

A[] + B[] -> C[]

On the CPU:

```
float *C = malloc(N * sizeof(float));
for (int i = 0; i < N; i++)
C[i] = A[i] + B[i];
return C;</pre>
```

On CPUs the above code operates <u>sequentially</u>, but can we do better, still on CPUs?

A simple problem...

On the CPU (<u>multi-threaded</u>, pseudocode):

```
(allocate memory for array C)
Create # of threads equal to number of <u>cores</u> on processor (around 2, 4, perhaps 8?)
(Indicate portions of arrays A, B, C to each thread...)
...

In each thread,
For (i from beginning region of thread)
C[i] <- A[i] + B[i]
//lots of waiting involved for memory reads, writes, ...
Wait for threads to <u>synchronize</u>...

This is slightly faster - 2-8x (can be slightly more with other tricks)
```

A simple problem...

- How many <u>threads</u> are available on the CPUs? How can the <u>performance</u> scale with thread count?
 - (Each CPU can generally have two threads).
- Context switching:
 - The action of switching which thread is being processed
 - High penalty on the CPU (main computer)
 - Not a big issue on the GPU

A simple problem...

On the GPU:

GPU: Strengths Revealed

- Emphasis on parallelism on GPUs means we have lots of cores
- This allows us to run many threads simultaneously with virtually no context switches



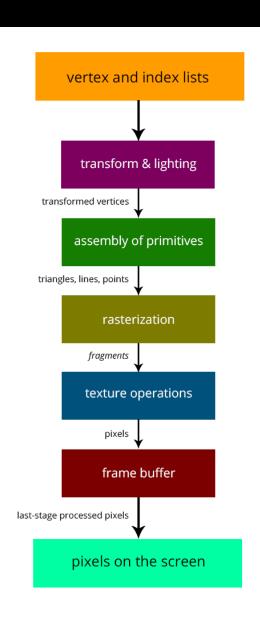
GPUs – Brief History

- Initially based on graphics focused fixed-function pipelines (<u>history</u>)
 - Pre-set <u>pixel/vertex functions</u>, limited options



http://game development.tutsplus.com/articles/the-end-of-fixed-function-rendering-pipelines-and-how-to-move-on-cms-21469

Source: Super Mario 64, by Nintendo



GPUs – Brief History

Shaders

- Can implement one's own functions using graphics routines.
- GLSL (C-like language), discussed in CS 171
- Can "sneak in" general-purpose programming! Uses pixel and vertex operations instead of general purpose code. Very crude.
- <u>Vulkan</u>/<u>OpenCL</u> is the modern multiplatform general purpose GPU compute system, but we won't be covering it in this course



http://minecraftsix.com/glsl-shaders-mod/

Using GPUs as "supercomputers"

"General-purpose computing on GPUs" (GPGPU)

 Hardware has gotten good enough to a point where it's basically having a mini-supercomputer

CUDA (Compute Unified Device Architecture)

General-purpose parallel computing platform for NVIDIA GPUs

Vulkan/OpenCL (Open Computing Language)

General heterogenous computing framework

Both are accessible as extensions to various languages

• If you're into python, checkout <u>Theano</u>, <u>pyCUDA</u>.

Upcoming GPU programming environment: Julia Language

GPU Computing: Step by Step

- Setup inputs on the host (CPU-accessible memory)
- Allocate memory for outputs on the host CPU
- Allocate memory for inputs on the GPU
- Allocate memory for outputs on the GPU
- Copy inputs from host to GPU (slow)
- Start GPU kernel (function that executes on gpu fast!)
- Copy output from GPU to host (slow)

NOTE: Copying can be asynchronous, <u>and unified memory</u> <u>management</u> is available

The Kernel

- This is our "parallel" function
- Given to each thread
- Simple example, implementation:

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c) {
    //Decide an index somehow
    c[index] = a[index] + b[index];
}
```

Indexing

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    c[index] = a[index] + b[index];
}
```

https://cs.calvin.edu/courses/cs/374/CUDA/CUDA-Thread-Indexing-Cheatsheet.pdf https://en.wikipedia.org/wiki/Thread_block

Calling the Kernel

```
void cudaAddVectors(const float* a, const float* b, float* c, size){
   //For now, suppose a and b were created before calling this function
   // dev a, dev b (for inputs) and dev c (for outputs) will be
    float * dev a;
    float * dev b;
    float * dev c;
   // Allocate memory on the GPU for our inputs:
   cudaMalloc((void **) &dev a, size*sizeof(float));
   cudaMemcpy(dev a, a, size*sizeof(float), cudaMemcpyHostToDevice);
   cudaMalloc((void **) &dev b, size*sizeof(float)); // and dev b
    cudaMemcpy(dev b, b, size*sizeof(float), cudaMemcpyHostToDevice);
   // Allocate memory on the GPu for our outputs:
   cudaMalloc((void **) &dev c, size*sizeof(float));
```

Calling the Kernel (2)

```
//At lowest, should be 32
//Limit of 512 (Tesla), 1024 (newer)
const unsigned int threadsPerBlock = 512;
//How many blocks we'll end up needing
const unsigned int blocks = ceil(size/float(threadsPerBlock));
//Call the kernel!
cudaAddVectorsKernel<<<blooks, threadsPerBlock>>>
    (dev a, dev b, dev c);
//Copy output from device to host (assume here that host memory
//for the output has been calculated)
cudaMemcpy(c, dev c, size*sizeof(float), cudaMemcpyDeviceToHost);
cudaFree(dev a);
cudaFree(dev b);
cudaFree(dev c);
```

- Solving PDEs on GPUs
- GPU vs CPU fluid mechanics
- Ray Traced Quaternion fractals and Julia Sets
- Deep Learning and GPUs
- Real-Time Signal Processing with GPUs

Questions can be live and interactive, on Zoom during office hours. Also can be posted on Piazza.

